JOURNAL OF SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
*J. Softw. Maint: Res. Pract.* **11**, 201–221 (1999)

**Research**

# Recovering Software Architecture from the Names of Source Files

NICOLAS ANQUETIL and TIMOTHY C. LETHBRIDGE*
*School of Information Technology and Engineering, University of Ottawa, Ottawa, Canada*

**SUMMARY**

**We discuss how to extract a useful set of subsystems from a set of existing source-code file names. This problem is challenging because many legacy systems use thousands of files names, including some that are very short and cryptic. At the same time the problem is important because software maintainers often find it difficult to understand such systems. We propose a general algorithm to cluster files based on their names, and a set of alternative methods for implementing the algorithm. One of the key tasks is picking candidate words to try to identify in file names. We do this by (a) iteratively decomposing file names, (b) finding common substrings, and (c) choosing words in routine names, in an English dictionary or in source-code comments. In addition, we investigate generating abbreviations from the candidate words in order to find matches in file names, as well as how to split file names into components given no word markers. To compare and evaluate our five approaches, we present two experiments. The first compares the 'concepts' found in each file name by each method with the results of manually decomposing file names. The second experiment compares automatically generated subsystems with subsystem examples proposed by experts. We conclude that two methods are most effective: extracting concepts using common substrings and extracting those concepts that relate to the names of routines in the files. Copyright © 1999 John Wiley & Sons, Ltd.**

KEY WORDS: design recovery; program understanding; concept extraction; name analysis; legacy systems; reverse engineering

## 1. INTRODUCTION

A common task presented to software maintainers is to try to understand a software system's architecture starting only from the set of files containing its source code. This task is frequently encountered when some change must be applied to a legacy system for which the original designers or implementors are no longer available.

In an ideal legacy system, one would find adequate documentation describing the requirements and the design of the well-organized source code. In fact, very few legacy systems conform to this idyllic vision. If at all available, documentation tends to be outdated, while repeated changes to the

*Correspondence to: Dr. Timothy C. Lethbridge, School of Information Technology and Engineering, 150 Louis Pasteur, University of Ottawa, P. O. Box 450, Stn. A, Ottawa K1N 6N5, Canada. Email: tcl@site.uottawa.ca
Contract/grant sponsor: NSERC
Contract/grant sponsor: Mitel Corporation
Contract/grant sponsor: Consortium for Software Engineering Research (CSER)

CCC 1040–550X/99/030201–21$17.50

*Received 7 December 1998*
*Revised 19 March 1999*

source code have blurred the original design. Furthermore, the source code itself is often organized as a very large set of files in a single directory. Older operating systems often force the following constraints on these files:

- They tend not to be organized into hierarchies of subdirectories, thus the single directory containing the source code can contain thousands of files.
- Their names tend to have a limited number of characters (perhaps 8 or 12) with few markers (e.g., underscore characters or initial uppercase letters) to divide names into components.

In a prior report on our work (Anquetil and Lethbridge, 1997) we explained how we were led to consider file names as a file clustering criterion. We worked on a legacy software system more than 15 years old with more than two million lines of code (LOC) and more than 1 800 different file names. We wanted to create a conceptual browser for the software maintainers of this software system. To assist our research, we asked the system's maintainers to give us examples of subsystems with which they were familiar. Studying each subsystem, it was obvious that its members displayed a strong similarity among their names. For each subsystem, concept names or concept abbreviations like 'q2000' (the name of a particular subsystem), 'list' (in a list manager subsystem) or 'cp' (for 'call processing') could be found in all the file names. Thus, we were prompted to use concepts embedded in file names as a file clustering criterion.

If our objective were not creating a browser, but instead were re-engineering, then a key requirement would be precise preservation of the semantics of the design. However, to assist in the software comprehension process does not require such precision. Thus, our informal file-name clustering criterion is a feasible approach to helping software maintainers understand the architecture of legacy systems that suffer from the two constraints noted above. If either constraint is lifted—as is the case in most modern software—the process of understanding software architecture is made considerably easier. We can see an immediate view of aspects of the architecture by looking at how the software is divided into directories. With long file names, the purpose of a file may be clear from its name.

In this paper, we explore the feasibility and efficiency of using concept abbreviations encoded in file names to gather these files in meaningful clusters (subsystems). The rest of this paper is organized as follows: in the next section (Section 2) we present the state-of-the-art in software architecture recovery and propose a new approach based on file name decomposition. In Section 3 we discuss some related work by ourselves and others. Then (Section 4), we propose algorithms that enable us to cluster files according to their file names. The clustering process has a number of interesting difficulties. In Section 5, we present a series of experiments we have conducted that illustrate the efficiency of the clustering. We close the paper (Section 6) with a discussion of our new approach.

## 2. A NEW APPROACH TO RECOVERING SOFTWARE ARCHITECTURE

### 2.1. State-of-the-art

We will first set the context of this research by presenting the state of the art in software architecture recovery. Then we propose a new approach based on file names and discuss some

 

possible outcomes of it. When a software maintainer sets out to understand a legacy system of the type described above, he or she may rely on several of the following sources of knowledge: the application domain, the principles of software engineering, the organization of the company, the software system itself, common sense, etc.

Automated approaches must rely on fewer sources of knowledge. Traditionally, the two main ones are:

- *Code clustering* (or the bottom–up approach). This approach consists of looking for regularities in the code (e.g., references to the same variable) and grouping pieces of code (files, routines, etc.) together according to these regularities. The clusters thus created are believed to form meaningful concepts about the design of the software as represented in the source code. This approach is based on knowledge about the system that is at a very low level of abstraction.
- *Plan recognition* (or the top–down approach). This approach assumes a given base of programming plans or concepts (e.g., traversal of a linked list, counter handling), and tries to find these in the code. This approach is based on software engineering knowledge as well as the same low abstraction level knowledge about the system (i.e., source code).

Both the top–down and bottom–up approaches have one common point: they consider the contents of the source code as the sole source of information about the system. The two approaches have opposite advantages and drawbacks: the top–down approach is slow and inefficient, and difficult to port when systems are large. The bottom–up approach is fast and efficient, it can handle large systems with millions of LOC, and is portable across a wide range of software systems. However, it too manipulates data at a very low level of abstraction and those data contain much noise. This makes it difficult to extract significant abstract concepts. It is therefore very hard for an automated tool to make appropriate decisions about which cluster many components should be allocated to.

Researchers have recognized the need for more human-orientated approaches and approaches which would capture higher level abstractions (Biggerstaff, Mitbander and Webster, 1994). The goal of the top–down approach is in fact, to allow the system to be viewed in terms of application domain concepts or software design concepts. So far, however, research on this approach (e.g. Palthepu, Greer and McCalla, 1997; Quilici, 1994; Reeves and Schlesinger, 1997; Wills, 1990; Woods and Quilici, 1996) has mostly concentrated on the recognition of low-level software design concepts (linked lists, counters, etc.), and has hardly considered application domain concepts. Another problem is that the approach has proved to be computation intensive and does not scale up well. Typical research has only involved a few thousand LOC. Also the approach requires a knowledge base of the concepts to be recognized, which supposes a large initial investment.

## 2.2. The file-name approach

In this paper, we explore an alternative to the above approaches to architecture recovery: the use of file names to cluster source files. This approach enjoys the following advantages:

- It is largely language independent; a good approach might be usable for any set of files.

 

- File names are concise, therefore the approach requires few resources and should scale up nicely.
- It extracts concepts from the file names which are often application domain concepts and therefore highly significant to the software maintainers. We see this approach as more geared toward application domain knowledge than even the top–down approach.

The file-name approach differs from the top–down and bottom–up approaches by shifting the focus from the contents of the source code to other less formal but more abstract sources of information. It is based on a different kind of knowledge: one can see file names as knowledge of the system at a higher abstraction level than source code, giving insights into application domain knowledge. Like the bottom–up approach, the file-name approach can be portable and fast enough to deal with large software systems (for instance, we are working with a legacy system containing two million LOC). But since it uses an informal source of information, explicitly intended for humans, it should also facilitate extracting concepts that are more 'human orientated'.

It is clear that clustering files according to their names will only provide us with a very coarse view of the software architecture. As Merlo, McAdam and De Mori (1993) state, 'many sources of information may need to be used during the (design) recovery process', and other approaches will be useful. File-name clustering is one solution that can give a new point of view on a system. Many other solutions can also be used, as for example, design recovery based on comment analysis, or on operational information such as provided by job control files. At the end of the paper, we will discuss some considerations for combining the various techniques.

For the particular legacy system we are studying, the file-name approach was the one that best matched the way the software maintainers viewed the system. We asked them to give us examples of small subsystems they knew. These subsystem examples were clearly identified by the occurrence of a concept in all the file names (Anquetil and Lethbridge, 1997).

## 2.3. What is in a name?

Before we attempt to analyse systems according to file naming conventions, we should give some thought to the forces that shape the names. Software maintainers and developers (including analysts, programmers, database administrators, etc.) can use any combination of the following as names for files, or as components of file names:

- Data manipulated (e.g., classes, structures, tables, ...). In the particular system we are studying, we found abbreviations such as: flag, db (database), str (string) or queue.
- Algorithms or processes (or steps of these) performed (these may be descriptive or mere symbols such as 'process1'). The file names we studied included abbreviations such as: mon (monitor), write, free, select, cnv (conversion) or chk (checking).
- Program control implemented (e.g., state machine, dispatcher, event handler, etc.). We found the following examples: svr (server), d (distributed), mgr (manager).
- The time period during which processing occurs, for example in our system: boot, ini (init), rt (runtime).
- I/O devices, services or external systems interacted with. In our telecommunication software one can find the following examples: k2 (sx2000, a particular product), sw (switch), f (fibre), alarm or kermit.

 

- Features implemented (a particular end-user function, or an entire sub-application). The following abbreviations were found: abrvdial (abbreviated dialling), mtce (maintenance), edit (editor).
- The names of other applications from where code was reused to create this system. We have no example of this particular category for the system we studied.
- The names of groups or programmers who developed the code. Again, this possibility was not used in our system.
- Versions of the files or software (e.g., the number 2 or the word 'new' may be added, or the name of target hardware). We may cite the abbreviations: s and sg (two different versions of a product in the company), na (North America), ma (Malaysia), etc.
- Problems that were fixed by adding a file. The system we studied had no examples.

The borders between these categories are not strict and often one abbreviation could fit in two categories. There are also abbreviations like 'utl' (utilities) which are difficult to categorize.

Many organizations have well-established naming conventions that may require some particular combination of the above elements to be used in file names. More often, however, the naming convention is developed informally. Indeed there may be multiple conventions employed by different groups and individuals. Despite the heterogeneity of naming conventions, it seems reasonable that if we can divide the system up according to almost any of the above criteria, we can obtain information that would be useful to the maintainer. A key consequence or limitation of using the file-name approach is that the resulting decomposition does depend heavily on semi-arbitrary choices made by the designers about which of the above criteria to use. In other words, the method might provide extremely useful information, but we will find different types of information from different systems.

As we will see later, however, a real problem with the file name approach is that elements of the names deriving from the above criteria are highly abbreviated. Thus, the second limitation of the approach is that we will normally not be able to extract all the information present in file names, and there will be some uncertainty in the process. Most of the details of the algorithms we present in this paper are attempts to address this limitation and minimize its impact.

## 3. RELATED RESEARCH

There has been a lot of research in the reverse engineering community concerning the clustering of files into subsystems. Our approach to file clustering consists of extracting from file names the concepts they contain. We then cluster together files that refer to the same concepts in their names. This results in a set of overlapping file clusters which correspond to important application domain concepts (e.g., call processing, in our telecommunication environment; Q2000, the name of a particular product; etc.) and software engineering concepts (e.g., database, test, debug, etc.)

Most prior research uses the bottom–up approach. This approach defines interrelationships between files, and then clusters together the files with strong interrelationships. Examples of such interrelationships include calls from routine to routine, uses in one file of variables or types defined in another, as well as inclusion of a particular file by two or more others. Examples of this approach are found in Müller *et al.* (1993) using variable uses and routine calls, Tzerpos and Holt (1997) using routine calls and file inclusion, or Mancoridis and Holt (1996) also using file inclusion. Additional

 

examples are in the special issue of the *Communications of the ACM* on reverse engineering (Waters and Chikofsky, 1994) and in Lakhotia's (1997) comparison of many research approaches.

However, this kind of approach is impeded by the very low level of information on which it is based. For example, Carmichael, Tzerpos and Holt (1995) report their difficulties in extracting correct design information from a reasonably well designed, medium sized (300 000 LOC) recently developed system. The experiment used inclusion between files to deduce subsystems, but it turned out that each file included many other files, with some of these file relationships crossing important architectural boundaries. Biggerstaff, Mitbander and Webster (1994) advocated a more human-orientated approach that would actually help the user to relate structures in the program to his 'human orientated conceptual knowledge'.

Researchers in plan recognition (the top–down approach) claim that this is what they are doing, but to our knowledge, this approach has never been used for file clustering, and it is not clear how this could be possible. Moreover, up to now, this approach has focused on finding 'software engineering clichés' because it does not consider application domain concepts.

Some researchers have used file names to extract or help extract subsystems (Neighbors, 1996; Tzerpos and Holt, 1997) and report good results, although they do not formally quantify these results. These researchers manually defined the abbreviations of interest and then looked for these abbreviations in the file names. Apparently, these researchers were only looking for one abbreviation in each file name, the one marking the subsystem to which the file belongs. We, on the other hand, are trying to automatically extract all abbreviations from the file names, whether they mark a subsystem or not. We are not looking for a partitioning of the system, but for a set of overlapping concepts.

Merlo, McAdam and De Mori (1993) have an approach very close to ours—they clustered files based on concepts extracted from comments and identifiers. Their approach used a neural network to discover the important concepts. Again it only aimed at discovering subsystems, and did not decompose the file names. One interest of this approach is that it is not solely based on the code.

The 1995 Working Conference on Reverse Engineering had a special track entitled 'Analysis of Non-Code Sources'. The three papers (Butler *et al.*, 1995; Leite and Cerqueira, 1995; Lutsky, 1995) are concerned with analysis of documentation to extract information about the design of the system. These researchers worked on a source of information at an abstraction level even higher than ours. Their purpose was more akin to scavenging documentation than redesigning the software system.

Our research was prompted by the way the software we study seems to be organized. Example subsystems given to us by software maintainers seem to follow an informal file naming convention. One could argue that the system we are studying is a very peculiar one, well organized with an unusually strict file naming convention. We do not think this is the case. We already mentioned Neighbors (1996) and Tzerpos and Holt (1997) who had similar experiences with other systems. Neighbors' experience seems particularly significant as he reports on data 'collected from three large scientific systems over a 12-year period.'

Also, it seems unlikely that companies can successfully maintain huge software systems for years with constantly renewed maintenance teams without relying on some kind of structuring technique. For our software system, it appears to be file naming conventions. We do not pretend it is the sole solution to file clustering, but it is one of many possibilities. Hierarchical directories is another commonly used approach (e.g., in the Linux project (Debian, 1998)).

 

# 4. DECOMPOSING FILE NAMES

## 4.1. General algorithm

In this section, we present general approaches to clustering files according to their names and the difficulties inherent in these approaches. In the subsections we discuss strategies for overcoming the difficulties. We call the components of a file name 'abbreviations' because they often consist of abbreviated forms of words. All file names containing a given abbreviation are clustered together. This cluster of file names along with the abbreviation itself, form a 'concept'.

The general algorithm we use for clustering files using their names is as follows:

*G.1.* Split each file name into its constituent abbreviations.
*G.2.* Create a cluster for each abbreviation found in step *G.1.*
*G.3.* For each file name, put it into each of the clusters that corresponds to its abbreviations.

This general algorithm has three important problems as noted below. Overcoming these problems is the main subject of this paper.

- *The word-marker problem*: the first difficulty arises from the lack of word markers (capital letters, underscore or hyphen characters) in the types of legacy software we are studying. This means that step *G.1* of the general algorithm becomes quite difficult. There may be several ways to split a file name. For example 'fsblock' could be decomposed into 'fsb-lock' or 'fs-block'.
- *The abbreviation problem*: the second difficulty arises from the limited number of characters available for file names in legacy software. This causes the concepts embedded in file names to be often represented as very cryptic abbreviations, and not as entire words. Since words can be abbreviated in multiple ways and several words can have the same abbreviation, our task of splitting file names into words (step *G.1*) becomes all the harder.
- *The overlapping problem*: the third difficulty arises from the overlapping of some of the abbreviations embedded in the file names. There may be several ways to interpret the string 'mnut' in 'actmnuts'. It could stand for 'MeNU' + 'T. . . ' or for 'MoNitor' + 'UTility'.

Not only do these three problems challenge an automated system, but they also challenge a human who tries to analyse file names manually. We spent more than five hours decomposing the names of about 220 files. We later used this manual work to help evaluate our automated approaches.

These three problems make step *G.1* very difficult. We propose augmenting the general algorithm to do it as follows:

*G.1.1.* Create a set of candidate abbreviations that we might expect to find in file names. We might obtain these candidate abbreviations from comments, identifiers, a dictionary, etc.
*G.1.2.* Choose the important abbreviations from this set.
*G.1.3.* Look in each file name for occurrences of any of the candidate abbreviations identified in step *G.1.1.*
*G.1.4.* Try to find the most probable decomposition for each file name.

In the following subsections, we discuss approaches to achieve these steps. Subsections 4.2 to 4.5 address step *G.1.1.* Subsections 4.6 and 4.7 address step *G.1.2.* Subsection 4.8 addresses step

 

*G.1.4*. Each approach is imperfect; we want to aim for a compromise approach that optimizes the following properties:

- *Precision*: the approach should only generate actual abbreviations.
- *Completion*: the approach should not miss any actual abbreviations.

To promote completion, we want to generate as many abbreviations as possible. This will mainly be handled in step *G.1.1* of the general algorithm. It is our experience that high completion is relatively easy to achieve. Overall, we will promote precision over completion.

To promote precision, we want to be conservative. We want to accept only the abbreviations that have a good chance of being real concepts. This means filtering out simple noise words. This also means picking a source of abbreviations that is not likely to contain concepts outside the application domain. This is mainly handled in step *G.1.2* of the general algorithm.

## 4.2. Candidate abbreviations from file names (iterative approach)

The following algorithm can be used to try to decompose a set of file names into their abbreviations in the absence of word markers:

*α.1*. Assume that file names of length 2, 3 or 4 are single abbreviations; i.e., the file name has only one component. Put each such file name in the set of candidate abbreviations and remove it from the set of file names to decompose.

*α.2*. **while** there is a file name $f$ not decomposed and prefixed by a candidate abbreviation *a* **do**

- *α.3*. Remove $f$ from the set of file name to decompose
- *α.4*. Let *fs* be the suffix of $f$ after removing the prefix *a*
- *α.5*. **if** (length(*fs*) $<$ 5 characters) **or** is_english_word(*fs*), **then** add *fs* to the set of candidate abbreviations
- *α.6*. **else** add *fs* to the set of 'file names' to decompose.

Although it would appear that this algorithm can tackle step *G.1* of the general algorithm (decomposing each file name into abbreviations), we use it only for step *G.1.1* (generating candidate abbreviations). The algorithm can only deal with about one fifth of the file names. Some of the abbreviations we extract with this algorithm do appear inside file names that are not decomposed here. This is because they are not prefixes of these file names (step *α.2*.). Because of the three problems mentioned in the previous subsection, relaxing this 'prefix' constraint would render the algorithm useless, i.e., decomposing the file names erroneously and extracting wrong abbreviations.

Even as described, the algorithm does make mistakes. For example 'activ' and 'activity' are two valid abbreviations. If we already asserted that 'activ' is an abbreviation, the algorithm will wrongly deduce that 'ity' is another one (see overlapping and abbreviation problems in the previous section). This kind of wrong abbreviation is dangerous since it may actually be found in other file names.

This algorithm is based on several assumptions, specific to our system, which may or may not be valid. Firstly, some four-character file names might be composed of two abbreviations (e.g., 'swid' stands for SoftWare IDentifier). Secondly, some file names longer than four characters might represent individual concepts that should not be decomposed ('kermit', 'activity', etc.). Despite these drawbacks, this algorithm does not seem to generate many wrong abbreviations, therefore it will not require a subsequent step *G.1.2*.

 

## 4.3. Candidate abbreviations from file names (statistical approach)

Another way of extracting candidate abbreviations from file names is to take substrings common to several file names. The high-level objective of the research is to allow software maintainers to browse clusters of files, where each cluster represents a concept that is implemented by the files of that cluster. This implies that we are not interested in concepts (abbreviations) that appear in only one file name. Conversely, we are primarily interested in abbreviations that appear in several file names.

Extracting all common substrings from a set of names may be very time and space consuming. A naive way to do it would consist of extracting all the substrings (of any length) of each name and then trying to find similar substrings in different names. However, in practice this can prove impossible because the number of substrings is quadratic in the length of the name.

We propose instead to extract from each name, all the substrings of a given length it contains. These strings are called N-grams (Kimbrell, 1988). For a length of 3, one speaks of 3-grams. For example, 'listdbg' contains the following 3-grams: 'lis', 'ist', 'std', 'tdb' and 'dbg'. The number of N-grams in a name is linear in the length of this name. Applying N-grams, we use the following algorithm:

*β.1.* Extract all the 3-grams from all the file names
*β.2.* Generate a Galois lattice (Godin and Mili, 1993; Wille, 1982). This structure clusters file names which share N-grams. It has an important property; it will find all clusters of file names sharing one or more N-gram(s).
*β.3.* **foreach** cluster *c* in the Galois lattice **do**
  *β.4.* Let *sstr* be the substring shared by the file names in *c*
  *β.5.* **if** (length(*sstr*) $< 5$ ) **or** is_english_word(*sstr*), **then** accept *sstr* as a candidate abbreviation
  *β.6.* **else** reject *sstr*.

Again, we are making important assumptions here: a candidate abbreviation is either an English word or shorter than five characters (step *β.5*). Note that the abbreviations cannot be less than three characters in this case because the algorithm is based on 3-gram extraction (step *β.1*). This is an important decision: if N-grams are too long, then short substrings will not be detected; if too short, then we will produce too many clusters to be manageable (or too many abbreviations). We tried to use 2-grams, but the algorithm extracted too many wrong abbreviations. Clearly, this $\beta$ method will tend to extract many abbreviations, including wrong ones. It will therefore require the filtering step *G.1.2.*

## 4.4. Candidate abbreviations from identifiers

We might also extract candidate abbreviations from the identifiers in the source code. In general, identifiers include the names of routines, variables, structured types, etc. However, in our experiments, we considered only routine names. We made this choice both for practical reasons and because we believed routines have a better chance of addressing the same concepts as the files.

Merlo, McAdam and De Mori (1993) already recognized routines as a useful file clustering criterion. Here, however, we do not want to cluster files based on the identifier names they contain (this has already proved unsuccessful in our case; see Anquetil and Lethbridge (1997)). Instead we

 

use identifiers as an easy way to get some abbreviations the software maintainers may use. In the system we are working with, identifier names are much easier to decompose than file names. They do not exhibit the word-marker problem, they make use of the underscore character and capital letters. As a consequence, the overlapping problem is not an issue either. Finally, they do not exhibit the abbreviation problem since there is no constraint on the length of identifiers.

There is a problem with many routine names that are prefixed by the name of the file that defines them. We do not want these file names to be unconditionally proposed as abbreviations, therefore we will again limit the length of abbreviations to less than five characters in our algorithm using identifiers:

$\gamma.1.$ **foreach** routine identifier $i$ **do**
  $\gamma.2.$ **foreach** 'word' $w$ in $i$ **do**
  $\gamma.3.$ **if** (length($w$) $< 5$ ) **or** is_english_word($w$), **then** accept $w$ as a candidate abbreviation
  $\gamma.4.$ **else** reject $w$.

## 4.5. Candidate abbreviations from English words

In several of the above methods, we noted that we would accept as abbreviations actual English words (e.g., 'list'). This gave us the idea of using an English dictionary to produce abbreviations, each word being a candidate abbreviation. This would obviously give poor results by itself, because many abbreviations are not English words, but it may be a good auxiliary technique when combined with other sources. The English dictionary we used is the one usually found in `/usr/dict/words` on Unix systems. On our computer system (a Solaris 2.5), it contains a little more than 25 000 words.

Using the dictionary introduces a precision problem. It contains words like 'in', 'the' or 'AC' that could easily be found in file names but would not be abbreviations in our system. We tried to use a standard stop word list (from the information retrieval system Smart (Buckley, 1998)); however, this proved harmful because the stop word list apparently contained important words. Some more specialized stop list should be set up. Also, this dictionary is not complete. It lacks some common words (e.g., 'display'), technical words (e.g., 'kermit') and also the ability to inflect words, such as to conjugate verbs or make nouns plural. A more intelligent tool like 'ispell' could give better results, but this tool could still not deal with the application domain concepts, or the abbreviated forms of words.

## 4.6. Filtering candidate abbreviations with comments

This subsection and the following pertain to step *G.1.2* of the general algorithm: how to filter out abbreviations that are not useful. It will only be applied in cases where we have first used the method described in Section 4.3, that is, the one generating excessive numbers of candidate abbreviations.

Many 'abbreviations' appear in the comments, whether they are English words or application domain terms. The filter simply consists of keeping only those candidate abbreviations we find in the comments. A similar method has already been used by Merlo, McAdam and De Mori (1993). They clustered files according to similar concepts found in the comments or in identifiers (see also

 

Section 4.4 above). In the software system we are studying, many files (70%) have a summary comment, which describes the main purpose of the file (as opposed to comments describing routines or details of the implementation). We restricted ourselves to these summary comments. The primary purpose of this restriction was to limit the size of data we have to deal with.

In a first experiment, we looked for a candidate abbreviation only in the summary comment of files that have an occurrence of that abbreviation in their names. In a second experiment, we used abbreviations found in the summary comments of all files; this gave better results and we will only present this method below. We believe that because comments are only used as a filter for pre-generated substrings, the more comments we have, the better it would be. As a logical conclusion, it seems that although the summary comments may appear more 'focused', we might get better results using all of them.

## 4.7. Filtering candidate abbreviations with abbreviation rules

The preceding filtering method works fine for abbreviations which are English words or application domain concepts, because we might expect to find these in the comments. But, in the file names, many words are actually abbreviated for lack of space (i.e., the abbreviation problem described earlier). For example, a word like 'debug' may appear in its full form in the comments but is abbreviated 'dbg' in the file names.

We propose the following algorithm to try to recognize these abbreviated forms:

*δ.1.* **foreach** candidate abbreviation *a*
  *δ.2.* **foreach** file name *f* containing an occurrence of *a*
    *δ.3.* **foreach** possible word *w* in the summary comment of *f*
      *δ.4.* **if** *a* is a prefix of *w* (e.g., 'aud' for 'audit'), **then** accept *a* as a candidate abbreviation
      *δ.5.* **elseif** *a* is composed of the first letter of *w* and all its consonants (e.g., 'abbrvt' for 'abbreviate'), **then** accept *a* as a candidate abbreviation
      *δ.6.* **else** reject *a*

This method is computation intensive. To make it practical, we had to limit the number of words we would consider as possible full versions of an abbreviation (step *δ.3*). This was done by only considering words inside the summary comments of files that contain an occurrence of the abbreviation in their name. Most of the abbreviations that are accepted by this algorithm are prefixes of words (step *δ.4*); the second rule rarely applies because it is too restrictive. For example, a common abbreviation for server is 'svr', where the first 'r' is dropped. It is our belief that relaxing this rule would be both inefficient in terms of time and results because a relaxed rule could accept too many erroneous abbreviations.

## 4.8. Generating splits

The following approach can be used to split a file name into abbreviations (step *G.1.4* of the general algorithm), when one has an initial list of candidate abbreviation. We call each decomposition of a name a 'split'. For each name, we find all candidate abbreviations it could

 

contain. We then generate all combinations of abbreviations that could compose the name. For more flexibility, we allow free characters in a split (a character not belonging to any abbreviation). For example, assuming we have two candidate abbreviations ‘list’ and ‘db’, the possible splits for ‘listdbg’ would be:

list – db – g
list – d – b – g
l – i – s – t – db – g
l – i – s – t – d – b – g

As there are relatively few possible splits for each name, we are able to generate all of them. But some are obviously wrong (e.g., the last one in the above example). We use a rating function to assess the correctness of the splits. We only keep the split(s) with the best rating (there may be ties). The current rating function simply gives a higher rating to a split with fewer members (candidate abbreviations and free characters). The rationale is to discard those splits with a lot of free characters.

We considered using more complex rating functions, with multiple criteria, such as:

- *Length of the abbreviations.* From informal study, it seems that three-character abbreviations are more numerous than other sizes. We could take this into account and give more weight to these abbreviations.
- *Number of sources proposing each abbreviation.* When using several sources together, we could give more weight to an abbreviation proposed by different sources.
- *Weighted source for abbreviations.* We do not have the same confidence in all sources. Some could have a higher weight than others, and therefore favour the abbreviations they generate.

The rating function does have an influence on the quality of the results (although it seems to be a minor one). But, because our goal is to compare various sources of abbreviations, we did not want to alter the results by using a rating function that is too complex.

## 5. EXPERIMENTS

### 5.1. Two comparisons

We have proposed a general algorithm to decompose file names. We have discussed several sources and methods of generating candidate abbreviations and have explained how to decompose file names using these candidate abbreviations. The results of these two experiments for the particular legacy software system we are studying will be presented in the next section on experimental results. In this section, we describe the two experiments we did to assess the efficiency of the different methods:

- Compare the splits from each method with a benchmark of manually decomposed file names.
- Compare the file clusters from each method with a benchmark of known subsystems.

 

## 5.2. Design of 'manual decomposition' experiment

For the first experiment, we manually decomposed a set of file names into their constituent concepts. For each file name in the benchmark, we automatically split (decomposed) it with all the methods described earlier and compared the results with the manual decomposition, counting how many abbreviations the automatic method found. To get a representative sample, the file names we decomposed to create the benchmark were the first 10 names (in alphabetical order) for each initial letter. As some of the letters have few or no file names beginning with them, we decomposed only 221 names. This sample represents 12% of the whole corpus (1 817 file names).

We decided to accept several correct splits for some names. There are different reasons for this:

- some words may be abbreviated in many ways ('activ' or 'act' for activity),
- some words are singular and plural ('error' and/or 'errors'),
- some abbreviations overlap ('saservice' stands for 'sas service'), and
- some words are composed ('wakeup').

In each of these cases, we did not want arbitrarily to choose one of the alternatives. We accept all 'sensible' splits. For example, for 'activity', we accept 'activity', 'activ' and 'act'. In the last two cases, the remainder of the word is simply ignored. In the case of overlapping abbreviations like 'saservice', we accept 'sa - service' and 'sas' ('ervice' we reject as an abbreviation and ignore it). For the 221 names, we accepted 256 splits. Only 31 names have more than one accepted split. The maximum is four accepted splits (for one name). Two names have three accepted splits (like 'activity') and 28 names have two accepted splits.

Using this benchmark, we measured the precision and recall of each decomposition method. Precision is the percentage of correct abbreviations in the generated split. Recall is the percentage of abbreviations extracted among all correct abbreviations (as defined in the benchmark). For example, in the benchmark 'listdbg' is decomposed as 'list - dbg'. If a method proposes the split 'list - db - g' it will get 33% precision and 50% recall. When the benchmark contains several valid splits for a name, or when a method extracts more than one best split (see the rating function described in Section 4.8), we take the result with the best precision. The final results for each method are the percentage recall and precision averaged over the 221 names.

## 5.3. Design of 'expert subsystems' experiment

As mentioned before, we aim at building a conceptual browser to help software maintainers navigate in a legacy software system. The 'quality' of the concepts extracted for this browser is evaluated by the first experiment. But, the research was prompted by subsystem examples given to us by software maintainers on the system. It seems interesting to evaluate how well we can replicate these example subsystems using file-name-based clustering.

This is not the same experiment as the first one, because not all the concepts embedded in a file name are used to mark subsystems. For example, the file 'listdbg' contains the abbreviations 'list' and 'dbg' (debug). According to the examples given to us, the first abbreviation does mark a subsystem (the 'list manager' subsystem), but the second one denotes a particular activity inside this subsystem—a collection of routines to help debug the subsystem's features.

To find the abbreviation in a file name that denotes the subsystem to which this file belongs, we use a simple heuristic that consists of keeping only the first abbreviation in the file name which is

 

not a single letter. This heuristic seems to match the organization of the system. This is the same heuristic used by Tzerpos and Holt (1997) for a different legacy system. We exclude those first abbreviations that are single letters to try to cope with such file names as 'qlistmgr' and 'flistaud' which are also members of the 'list' subsystem.

There are other more complex cases, which we cannot deal with using this heuristic. For example, one of the subsystem examples (the 'hotel/motel' subsystem) contains two exceptions out of eight file names: 'cphotel' and 'dbthotel'. These exceptions are complex cases where two subsystems are indicated in the file name ('cp' stands for 'call processing', a very large subsystem, and 'db' is a database subsystem).

The second difficulty of the experiment is how to compare the generated file clusters with the known subsystems. Each subsystem has a name (a label) and we could just try to count how many files a given method successfully assigns to the right label. But this could underestimate the result of the methods. Consider a method that correctly groups the files with the exception that it creates file clusters smaller than the subsystems (each subsystem might be cut in two halves for example). We would like this method to get a relatively good score because it is clustering the files correctly, only it extracts clusters at a slightly lower level of abstraction than the subsystem examples we have. What we really want to measure is whether the method clustered together the files that are related.

The experiment consists of comparing pairs of files as classified by a method and as classified by the expert. Two given files may be in the same subsystem (we call it an 'intra-pair') or in different subsystems (an 'inter-pair'). We say that a method has a high recall if many of the intra-pairs of the subsystem examples are also intra-pairs for the method, i.e., if the method does not separate the experts' examples. Note that this is a different recall metric than in the previous section. A simple but pointless way to have a good recall would be to have only one huge cluster.

We will say that a method has a high precision if many of its intra-pairs are also intra-pairs in the experts' examples, i.e., if the method does not group files which should be separate. Again, this is a different precision metric than in the previous section. A simple way to generate high precision would be to have no clusters at all (therefore 'all zero' of the grouped pairs would be found in the example subsystems).

## 6. EXPERIMENTAL RESULTS

### 6.1. Results of 'manual decomposition' experiment

The results for this experiment are presented in Figure 1 and Table 1. We give in Figure 1 the efficiency of each method in terms of precision and recall. Table 1 gives the number of extracted candidate abbreviations which are not English words. We will first analyze the results of the four methods for generating candidate abbreviations (step *G.1.1*).

*Dictionary*: this method gives poor results, however we mentioned it was not intended to be used alone but only as an improvement when combined with other methods. For all other methods, we give the efficiency of the method alone and when combined with the English dictionary.

*File names (iterative method)*: the method alone does not perform well (40% precision, 50% recall). However, when combined with the English dictionary it scores much better. One could think the bad results might be due to the low number of proposed abbreviations (see Table 1). But, we

 

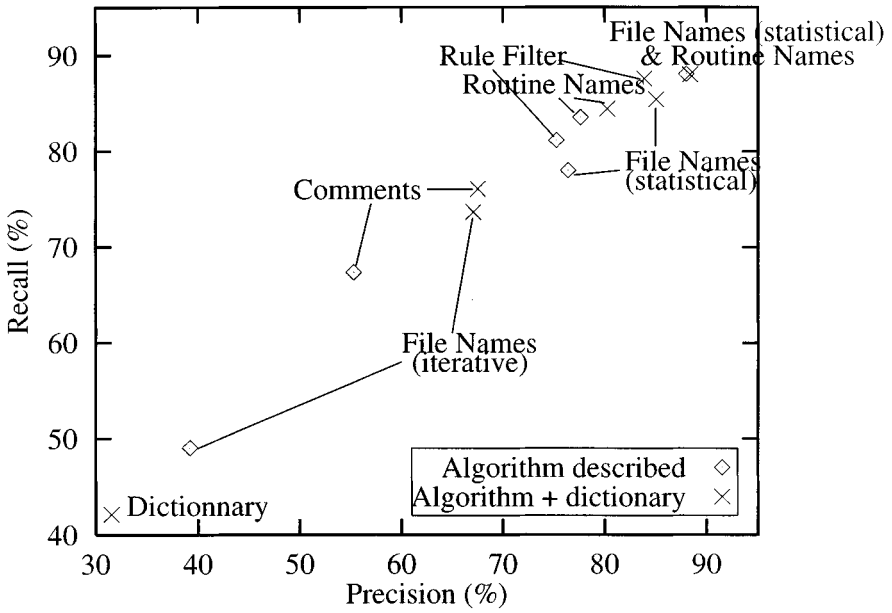


*Figure 1. Efficiency of file name decomposition methods*

will see that another method, using the 'comments' filter, proposes only slightly more abbreviations and gives much better results.

*File names (statistical method)*: the method works significantly better than the iterative method, and is still improved by the addition of the dictionary. Note that this method alone proposes all substrings common to more than one file name, therefore, the improvement introduced by the dictionary should be ascribed to words that are used in a single file name. This reduces their interest since they will not help us to cluster files together. They may, nevertheless, have a beneficial influence on step *G.1.4* (decomposition of the file names using all candidate abbreviations).

*Routine names*: this is the best of the candidate abbreviation generation methods, when each method is used alone. Combining it with use of dictionary brings an improvement, but less noticeable. It seems natural that with better and better results, it becomes more and more difficult to get significant improvements. One may notice that although the routine method alone scores better than the file name statistical method alone, with the addition of the dictionary, the order is reversed. One may suppose that since routine names are less constrained in their length than file names, the routine method already proposes many valid words, and therefore benefits less from the dictionary. This seems to be confirmed by Table 1.

*Filtering with comments*: the filtering method results shown here were applied on the file name statistical method. We also tried to filter the routine names method, but the results were slightly worse. This filter does eliminate many candidate abbreviations (mainly non-English words). Unfortunately, this is done at the expense of the method's efficiency.

*Filtering with comments and rules*: a less strict filter accepts abbreviations found in the comments that can be generated from words in comments by some simple abbreviation rules. This filter is more

 

Table 1. Number of English words and non-words proposed by each method

| Method of obtaining words | English words | Non-English words |
|---|---|---|
| File names (iterative) | 81 | 497 |
| File names (statistical) | 238 | 1 872 |
| Routine names | 1 132 | 2 111 |
| Dictionary | 25 143 | 0 |
| Comments | 130 | 368 |
| Rules filter | 181 | 762 |
| Routines and file names (statistical) | 1 208 | 3 475 |

permissive than the previous one. For example, it accepts twice as many non-English words. The results are also surprisingly good, better in fact than the abbreviation generation method (file name statistical) itself, whereas it still proposes many fewer candidate abbreviations.

We believe that the improvement comes from the last stage of the general algorithm (step *G.1.4*). Having a smaller set of candidate abbreviations, and hopefully a more precise one, this step is better able to decompose the file names. This seems to support the analysis we made: comments are useful to extract full words and application domain concepts, whereas computing abbreviated forms will provide the other abbreviations.

*A combination of file names (statistical) and routine names*: finally, we combined the two best methods for generating abbreviations: file name statistical and routine names. The results are extremely good (precision and recall close to 90%). Adding the dictionary does not bring noticeable improvement, which seems understandable at these already high scores.

## 6.2. Results of 'expert subsystems' experiment

The second experiment gives slightly different results. However, keep in mind that it is based on example subsystems that are not fully representative of the system because they contain a small number of files (4% of the entire set).

In Figure 2 we give the results of the second experiment in terms of the precision and recall of each method regarding the inter and intra-pairs they extract. Table 2 gives the average size of the subsystems extracted by each method.

Note that precision is in all cases very high (>90%). This means that the methods do not group unrelated file names. A possible explanation is that the methods tend to generate smaller subsystems than those of the experts, which favours the precision. Compare the average size of the experts' examples (6.8 files per subsystem) with the numbers in Table 2. This lower average size could be the cause of the general high precision, although there is no direct link between the two. For example, adding the dictionary to any method tends to decrease the average size, but in many cases (comments, rule, file names statistical, etc.), it also decreases the precision.

Dictionary gives bad results, which is expected given the previous experiment and also the fact that it generates clusters of a very low average size. The next result is that of the file names (iterative) method, which is still compatible with the previous experiment. The five other methods are all very

 

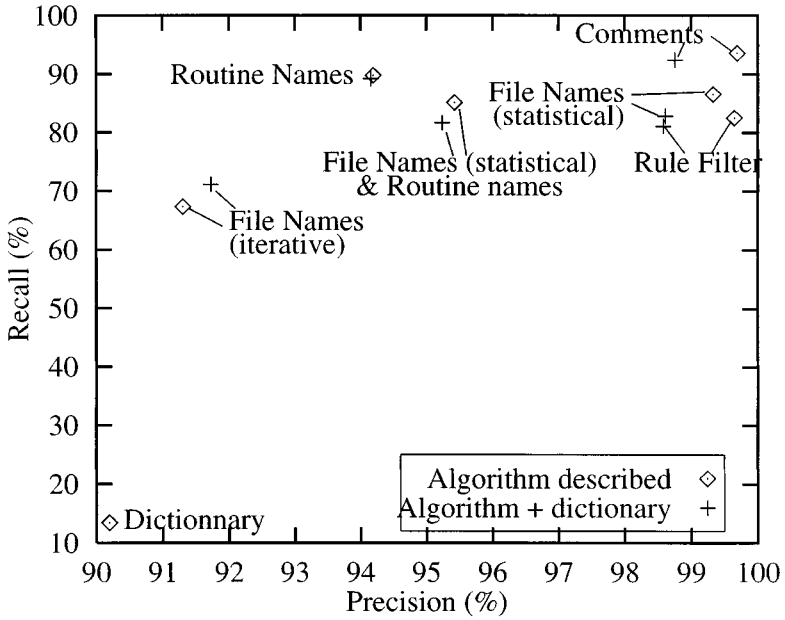


*Figure 2. Comparison of subsystems extracted with expert subsystem examples*

Table 2. Average size of the subsystems extracted by each method

| Method of obtaining words | Average size (alone) | Average size w/ dictionary |
|---|---|---|
| File names (iterative) | 5.79 | 4.43 |
| File names (statistical) | 3.13 | 3.05 |
| Routine names | 4.16 | 4.06 |
| Dictionary | 2.26 | N/A |
| Comments | 5.52 | 4.67 |
| Rules filter | 4.06 | 3.87 |
| Routines and file names (statistical) | 2.93 | 2.90 |

close (94% $\leq$ precision $\leq$ 100% and 80% $\leq$ recall $\leq$ 95%). The comments method has very good results. The recall may be improved by the relatively high average size of clusters, but one might have expected the precision to be adversely affected for the same reason.

In general, combined methods give worse results in this experiment, whereas they tended to perform better in the previous experiment. The file names (statistical) with routine names method which was the best method in the previous experiment, is now roughly in between its two components and in fact is worse than both of them regarding recall rate. Also, as we have already noted, adding the dictionary tends to give worse results (except for the file names (iterative) method).

 

# 7. DISCUSSION

## 7.1. Strategy and experiments presented

Discovering subsystems in a legacy software system is an essential but difficult practical task that raises important research questions. While studying a legacy telecommunication software system, and its software maintainers, we discovered that their definition of what constitutes a subsystem is mainly based on the files' names. This goes against the commonly accepted idea in the reverse-engineering research community that the body of the source code is the sole reliable source of information when performing file clustering.

Given these observations, our objective was to build a tool that uses file naming conventions to help software maintainers browse the concepts in a software system. We believe extracting concepts from names could greatly contribute to the design recovery activity by providing information of a higher abstraction level than source code. To discover the concepts represented by file names we need to extract the constituent abbreviations of the names. This is a difficult task that requires both a lot of knowledge to find the concepts to which file names refer, and a lot of intuition to associate abbreviated forms with the concepts.

In this paper we describe several methods of extracting abbreviations, and present experiments to compare them. The overall strategy we chose seems appropriate, as it allows us to get good results (90% precision and recall). The main steps of the algorithm are: (1) generate candidate abbreviations, and (2) generate a probable decomposition (split) of each file name given these abbreviations. An additional filtering step may be added in between these two steps.

From all the methods we propose, two achieved very good results in our first experiment and should prove difficult to improve upon:

- Extracting abbreviations from routine names that we will then look for in the file names. The great appeal of this method is its extreme simplicity.
- Extracting substrings that are common to a set of file names. This is the best of all, but it may be difficult to compute for large software systems.

In a second experiment, we tried to build subsystems using the file name decomposition and a simple heuristic to determine which abbreviation marks the subsystem to which the file belongs. However, because the set of real subsystem examples we have is small, the results may not be completely significant. As with the first experiment, the results are very good ($>$90% recall and quasi-perfect precision for three of the methods). However, the conclusions about which methods are best do not match the previous experiment. We propose an explanation that considers another factor: the average size of the subsystems.

We believe that the main interest of this research lies in the proof that informal sources of information may be used to help cluster files in subsystems. This departs from the traditional approach, which only considers the code.

## 7.2. Considerations for practical application

We intend the work presented in this paper to be incorporated into CASE tools. Currently, CASE tools provide a variety of design recovery approaches, but we know of no tool that does the kind of detailed analysis of file names that we have presented.

 

Clearly a CASE tool would want to combine this technique with others. Our technique can extract useful high-level information, but as mentioned in the introduction, the exact type of information depends on naming decisions made by designers and others, as well as the quality of those decisions. Other sources of information should also be used to give the maintainer the best possible decompositions. A CASE tool could present the user with several alternative decompositions of the system: e.g., a file-name decomposition as described here, one that uses analysis of file inclusions, plus another that analyses common references to routines, global variables, types etc. The user could then select the decomposition that he or she understands best.

Alternatively, the CASE tool could incorporate an algorithm that combines the results obtained from several different techniques (Anquetil and Lethbridge, 1999). Finally, the tool could use one technique to break the system into large subsystems, and then use another technique to divide it more finely.

### 7.3. Future work

The following are three suggestions for future research that build on the work presented in this paper:

- There are many possible avenues for researching how to combine this technique with others, as described in the last section.
- More work needs to be done regarding the abbreviated forms of words. Some words accept different abbreviations (e.g., monitor may be ‘mon’ or ‘mn’; activity may be ‘activ’ or ‘act’). This is what we called the ‘abbreviation problem’. Now that we are able to decompose correctly most of the file names, we could try to cluster together these related abbreviations. But, this appears to be a very difficult and knowledge-intensive task.
- Work could be done to design a method that would combine the properties of our approach (based on file names) and more traditional approaches (top–down or bottom–up). This would possibly mean that we should first quantify precisely the advantages of each approach.

## 8. CONCLUSION

The main contributions of this paper are:

1. a set of algorithms that use file names to cluster source files based on their names, and
2. experimental results that show the feasibility and usefulness of the approach to the reverse engineering of legacy software.

We believe that informal and high-level sources of information such as file names can provide more ‘human orientated’ decompositions of large sets of files, when compared with approaches that instead analyse low-level details of the source code. Maintainers will hopefully be able to use our methods to map parts of the system to application domain concepts or high-level design concepts.

## Acknowledgements

The authors thank the software maintainers at Mitel who participated in this research by providing us with data to study and by discussing ideas with us. We are also indebted to all the members of the KBRE

 

research group, especially Jelber Sayyad-Shirabad, Janice Singer and Stéphane Somé for fruitful discussions. The project URL is: http://www.site.uottawa.ca/~tcl/kbre

## References

Anquetil, N. and Lethbridge, T. C. (1997) 'File clustering using naming conventions for legacy systems', in Johnson, J. H. (Ed), *CASCON'97*, IBM Centre for Advanced Studies, Toronto, Canada, pp. 184–195.

Anquetil, N. and Lethbridge, T. C. (1999) *Combination of different clustering algorithms for reverse engineering*, Technical Report TR-99-02, Computer Science Department, University of Ottawa, Ottawa, Canada, 12 pp,

Buckley, C. (1998) *Smart v11.0*, available via anonymous ftp from: ftp.cs.cornell.edu, in pub/smart/smart.

Butler, G., Grogono, P., Shinghal, R. and Tjandra, I. (1995) 'Retrieving information from data flow diagrams', in *Proceedings Working Conference on Reverse Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 22–29.

Biggerstaff, T. J., Mitbander, B. J. and Webster, D. (1994) 'Program understanding and the concept assignment problem', *Communications of the ACM*, **37**(5), 72–83.

Carmichael, I., Tzerpos, V. and Holt, R. C. (1995) 'Design maintenance: unexpected architectural interactions', in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society, Los Alamitos CA, pp. 134–137.

Debian (1998) 'DEBIAN', Gnu/Linux web page URL: http://www.debian.org.

Leite, J. C. S. do P. and Cerqueira, P. M. (1995) 'Recovering business rules from structured analysis specification', in *Proceedings Working Conference on Reverse Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 13–21.

Godin, R. and Mili, H. (1993) 'Building and maintaining analysis-level class hierarchies using Galois lattices', *ACM SIGplan Notices*, **28**(10), 394–410.

Kimbrell, R. E. (1988) 'Searching for text? send an N-gram!', *Byte*, **13**(5), 297–312.

Lakhotia, A. (1997) 'A unified framework for expressing software subsystem classification techniques', *Journal of Systems and Software*, **36**(3), 211–231.

Lutsky, P. (1995) 'Automatic testing by reverse engineering of software documentation', in *Proceedings Working Conference on Reverse Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 8–12.

Mancoridis, S. and Holt, R. C. (1996) 'Recovering the structure of software systems using tube graph interconnection clustering', in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 23–32.

Merlo, E., McAdam, I. and De Mori, R. (1993) 'Source code informal information analysis using connectionist models', in Bajcsy, R. (Ed), *International Joint Conference on Artificial Intelligence, IJCAI'93*, volume 2, Morgan Kaufmann Publishers, Inc., San Francisco CA, pp. 1339–1344.

Müller, H. A., Orgun, M. A., Tilley, S. R. and Uhl, J. S. (1993) 'A reverse-engineering approach to subsystem structure identification', *Journal of Software Maintenance: Research and Practice*, **5**(4), 181–204.

Neighbors, J. (1996) 'Finding reusable software components in large systems', in *Proceedings Working Conference on Reverse Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 2–10.

Palthepu, S., Greer, J. E. and McCalla, G. I. (1997) 'Cliché recognition in legacy software: a scalable, knowledge-based approach', in *Proceedings Working Conference on Reverse Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 94–103.

Quilici, A. (1994) 'A memory-based approach to recognizing programming plans', *Communications of the ACM*, **37**(5), 84–93.

Reeves, A. A. and Schlesinger, J. D. (1997) 'Jackal: a hierarchical approach to program understanding', in *Proceedings Working Conference on Reverse Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 84–93.

Tzerpos, V. and Holt, R. C. (1997) 'The orphan adoption problem in architecture maintenance', in Verhoef, C., Baxter, I. and Quilici, A. (Eds) *Proceedings Working Conference on Reverse Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 76–82.

Waters, R. C. and Chikofsky, E. J. (Eds) (1994) 'Special issue on reverse engineering', *Communications of the ACM*, **37**(5).

 

Wille, R. (1982) 'Restructuring lattice theory: an approach based on hierarchies of concepts', in Rival, I. (Ed), *Ordered Sets: Proceedings of the NATO Advanced Studies Institute*, D. Reidel Publishing Company, Dordrecht, The Netherlands, pp. 445–470.

Wills, L. M. (1990) 'Automated program recognition: a feasibility demonstration', *Artificial Intelligence*, **45**(2), 113–171.

Woods, S. and Quilici, A. (1996) 'Some experiments toward understanding how program plan recognition algorithms scale', in *Proceedings Working Conference on Reverse Engineering*, IEEE Computer Society Press, Los Alamitos CA, pp. 21–30.

**Authors' biographies:**

**Nicolas Anquetil** is currently a Research Associate and part-time Professor at the University of Ottawa. His research interests include reverse engineering, knowledge-based approaches to software engineering, classification and the theoretical foundations of the object model. Nicolas received a D.E.A. in Computer Science (artificial intelligence) from the University of Caen in France in 1988. He completed his Ph.D. in Computer Science at the University of Montréal in 1996. His email address is: anquetil@site.uottawa.ca

**Timothy C. Lethbridge** is currently an Assistant Professor at the University of Ottawa, where his research interests include software design techniques, human–computer interaction, software engineering education and experimental software engineering. Previously he worked for the Government of New Brunswick and BNR (now Nortel Networks) on data processing and engineering software. Timothy received his Ph.D. from the University of Ottawa in 1994. His email address is: tcl@site.uottawa.ca

 